Pipelining and Hazards

CS 3410, Spring 2014

Computer Science
Cornell University

See P&H Chapter: 4.6-4.8

Announcements

Prelim next week

- Tuesday at 7:30.
- Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
- Go based on netid

Prelim reviews

- Friday and Sunday evening. 7:30 again.
- Location: TBA on piazza

Prelim conflicts

- Contact KB , Prof. Weatherspoon, Andrew Hirsch

Survey

- Constructive feedback is very welcome

Administrivia

Prelim1:

- Time: We will start at 7:30pm sharp, so come early
- Loc: Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
- Closed Book
  - Cannot use electronic device or outside material
- Practice prelims are online in CMS

- Material covered everything up to end of this week
  - Everything up to and including data hazards
  - Appendix B (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non] MIPS processor with hazards)
  - Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
  - Chapter 1 (Performance)
  - HW1, Lab0, Lab1, Lab2

Pipelining

Principle:

Throughput increased by parallel execution

Balanced pipeline very important

Else slowest stage dominates performance

Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards* (this and next lecture)

Basic Pipeline

Five stage "RISC" load-store architecture

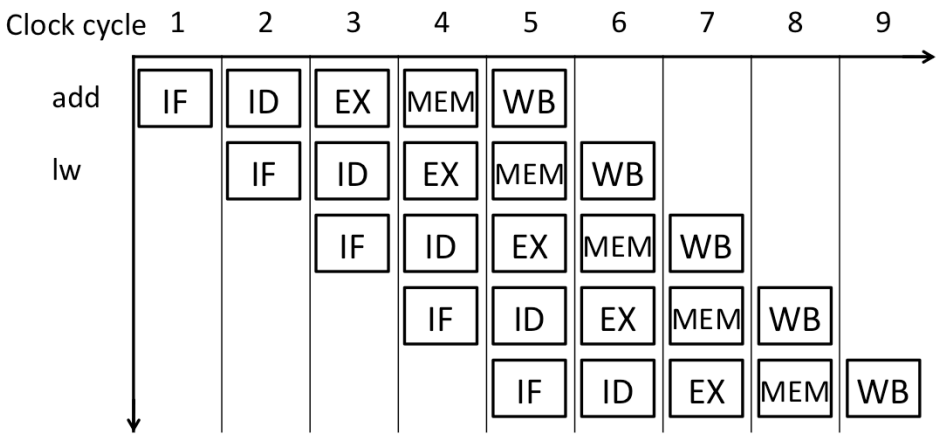
1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file

This is simpler than the MIPS, but we're using it to get the concepts across – everything you see here applies to MIPS, but we have to deal w/ fewer bits in these examples (that's why I like them)

Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time

Time Graphs



CPI = 1 (inverse of throughput)

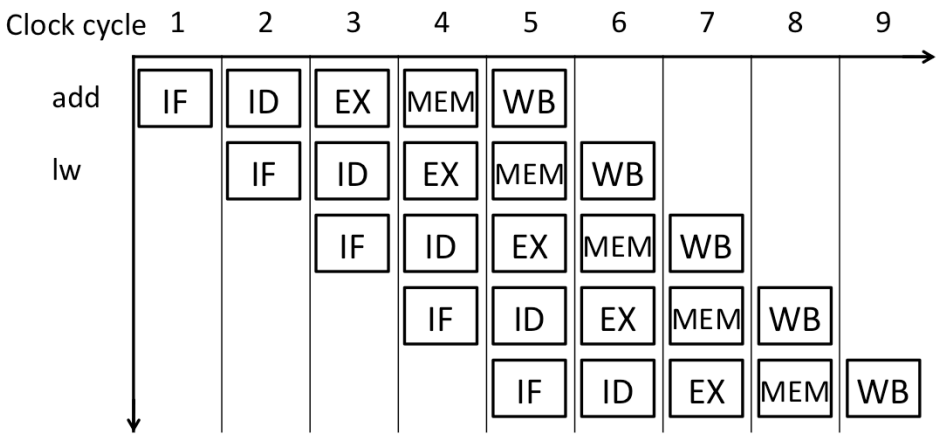
iClicker

The pipeline achieves

A) Latency: 1, throughput: 1 instr/cycle
B) Latency: 5, throughput: 1 instr/cycle
C) Latency: 1, throughput: 1/5 instr/cycle
D) Latency: 5, throughput: 5 instr/cycle
E) None of the above

Answer: B

Time Graphs



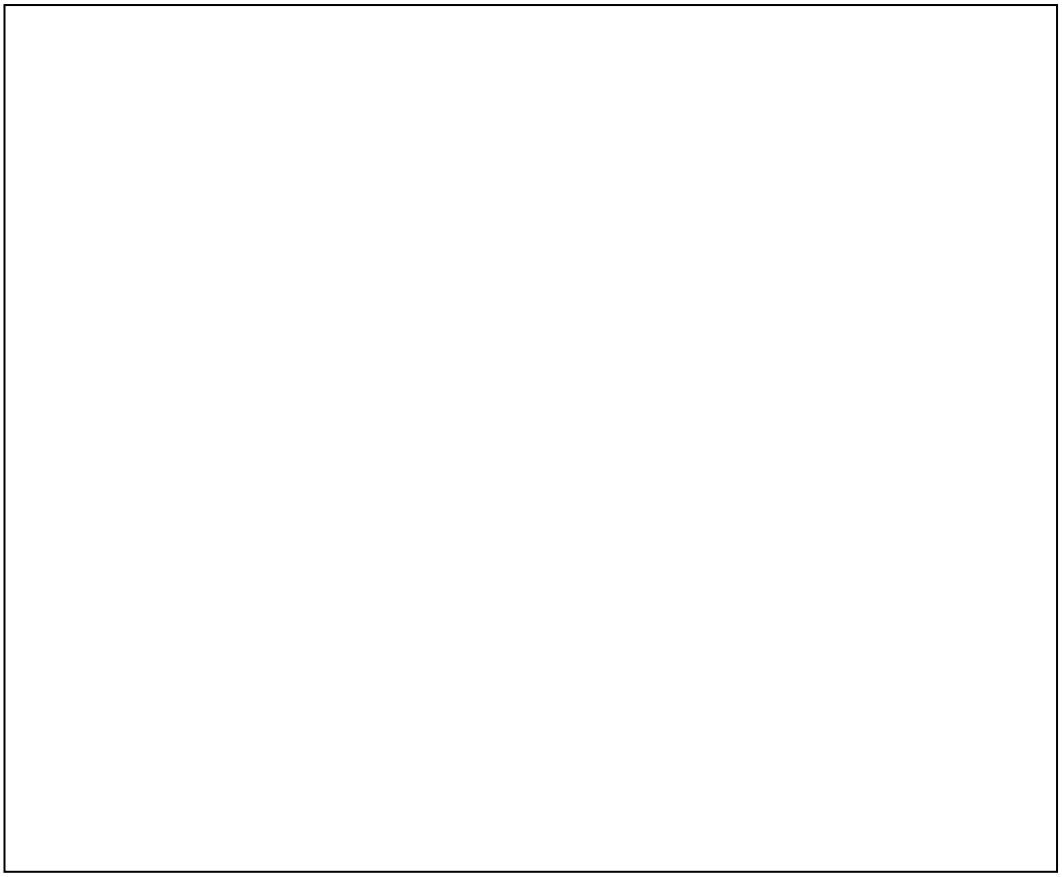
Latency: 5
Throughput: 1 instruction/cycle
Concurrency: 5

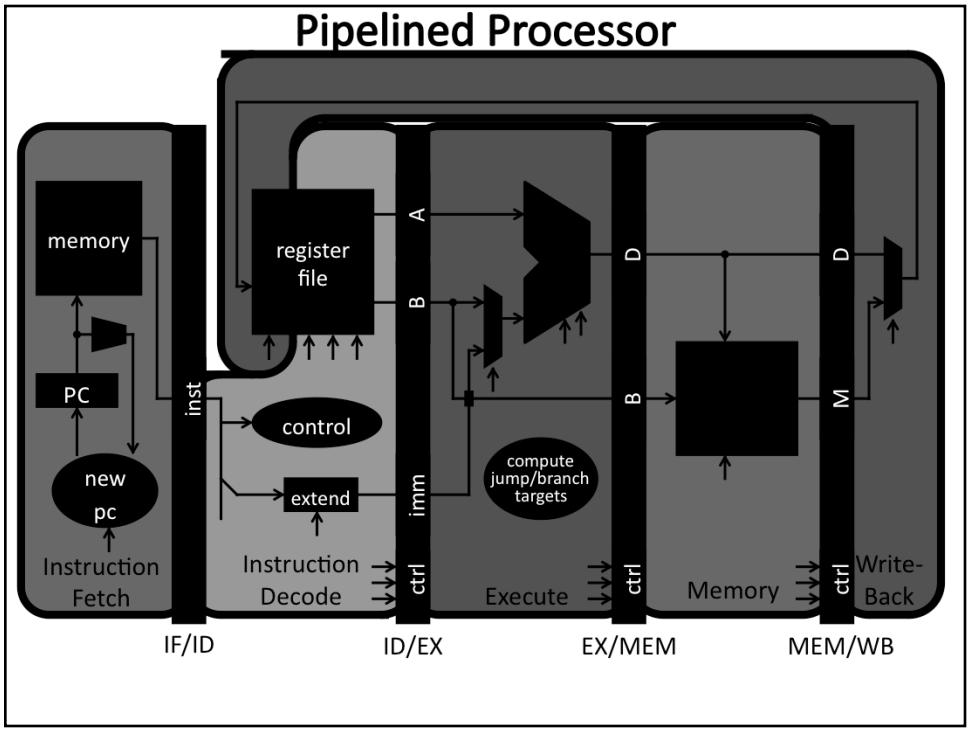
CPI = 1 (inverse of throughput)

Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages



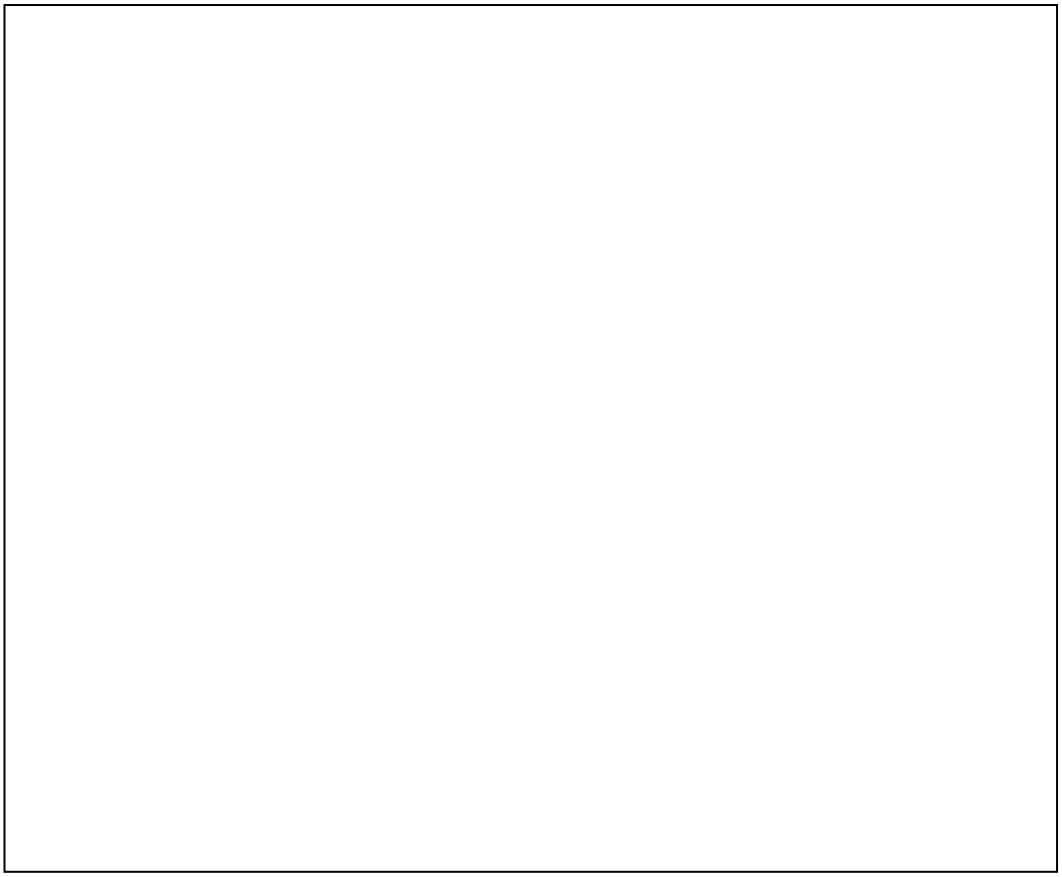

Pipelined Processor
memory
register file
PC
new pc
inst
control
extend
A
B
imm
ctrl
D
B
ctrl
compute jump/branch targets
D
M
ctrl
Instruction Fetch
Instruction Decode
Execute
Memory
Write-Back
IF/ID
ID/EX
EX/MEM
MEM/WB

Pipelined Implementation

- Each instruction goes through the 5 stages
  - Each stage takes one clock cycle
    - So slowest stage determines clock cycle time

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages

And is this it? Not quite….



Hazards

3 kinds

- Structural hazards
  - Multiple instructions want to use same unit
- Data hazards
  - Results of instruction needed before ready
- Control hazards
  - Don't know which side of branch to take

Will get back to this

First, how to pipeline when no hazards

1) Structural: say only 1 memory for instruction read and the MEM stage. That is a structural hazard. We have designed the MIPS ISA and our implementation of separate memory for instruction and data to avoid this problem. MIPS is designed very carefully to not have any structural hazards.

2) Data hazards arise when data needed by one instruction has not yet been computed (because it is further down the pipeline). We need a solution for this.

3) Control hazards arise when we don't know what the PC will be. This makes it not possible to push the next instruction down the pipeline. Till we know what the issue is.

IF

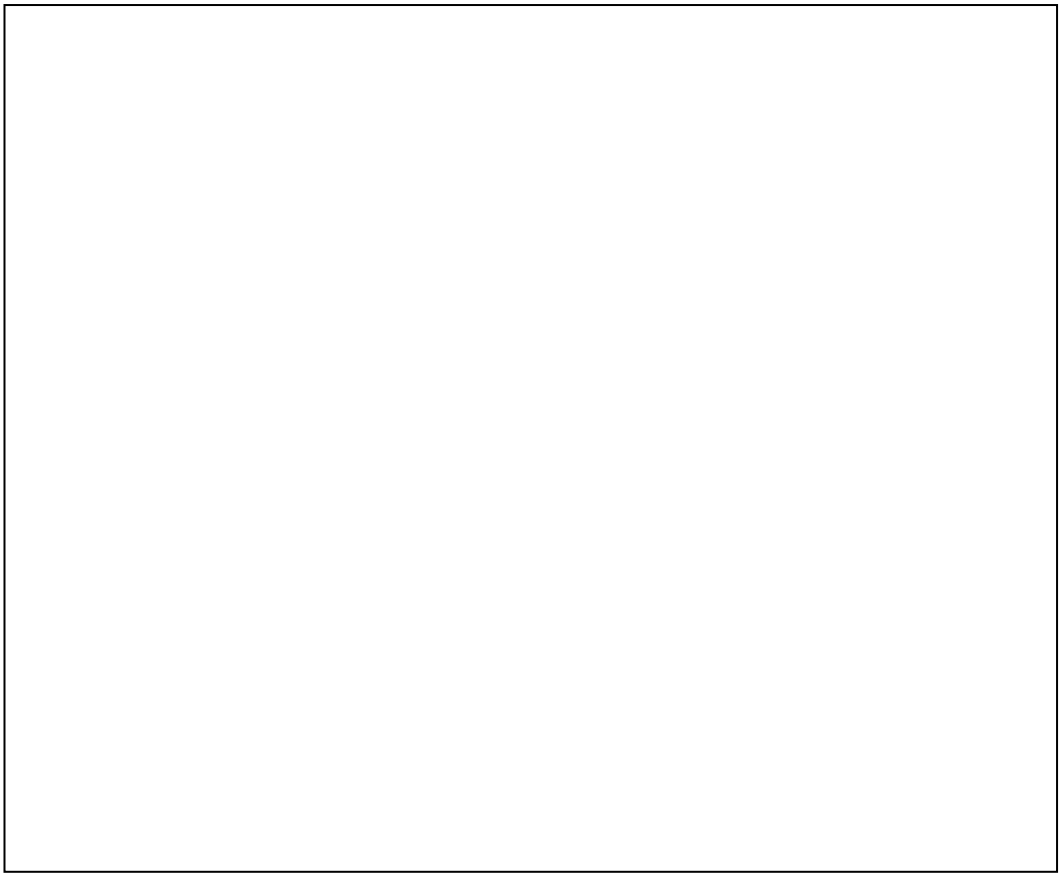
Stage 1: Instruction Fetch

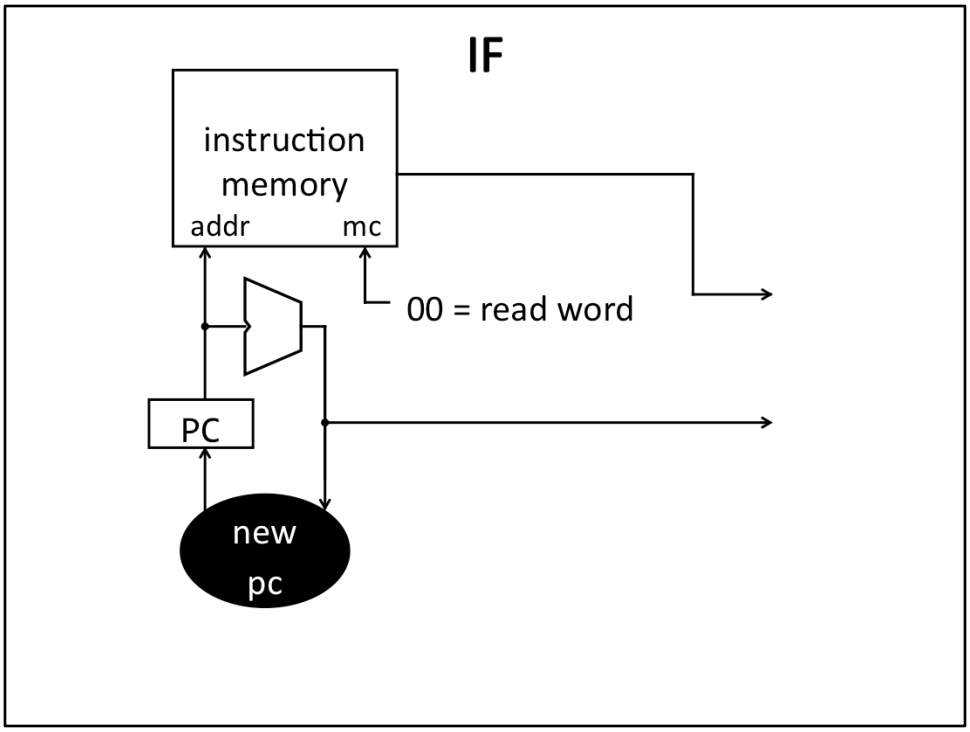
Fetch a new instruction every cycle

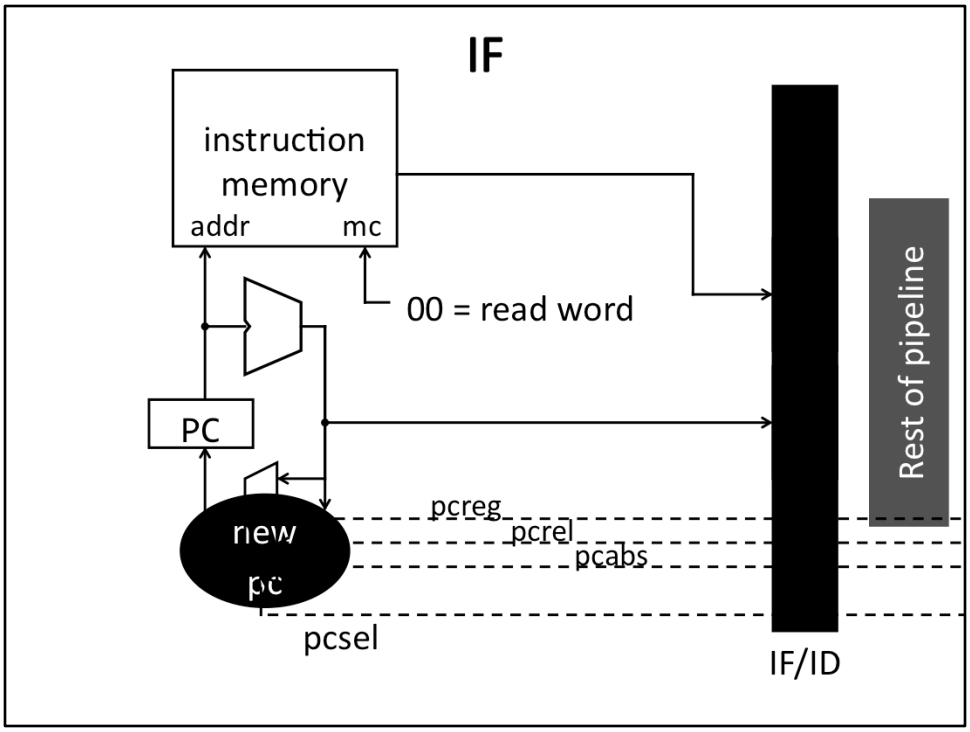
- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

Write values of interest to pipeline register (IF/ID)

- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)




IF
instruction
memory
addr
mc
00 = read word
PC
new
pc


IF
instruction
memory
addr
mc
00 = read word
PC
new
pc
pcreg
pcrel
pcabs
pcsel
IF/ID
Rest of pipeline

ID

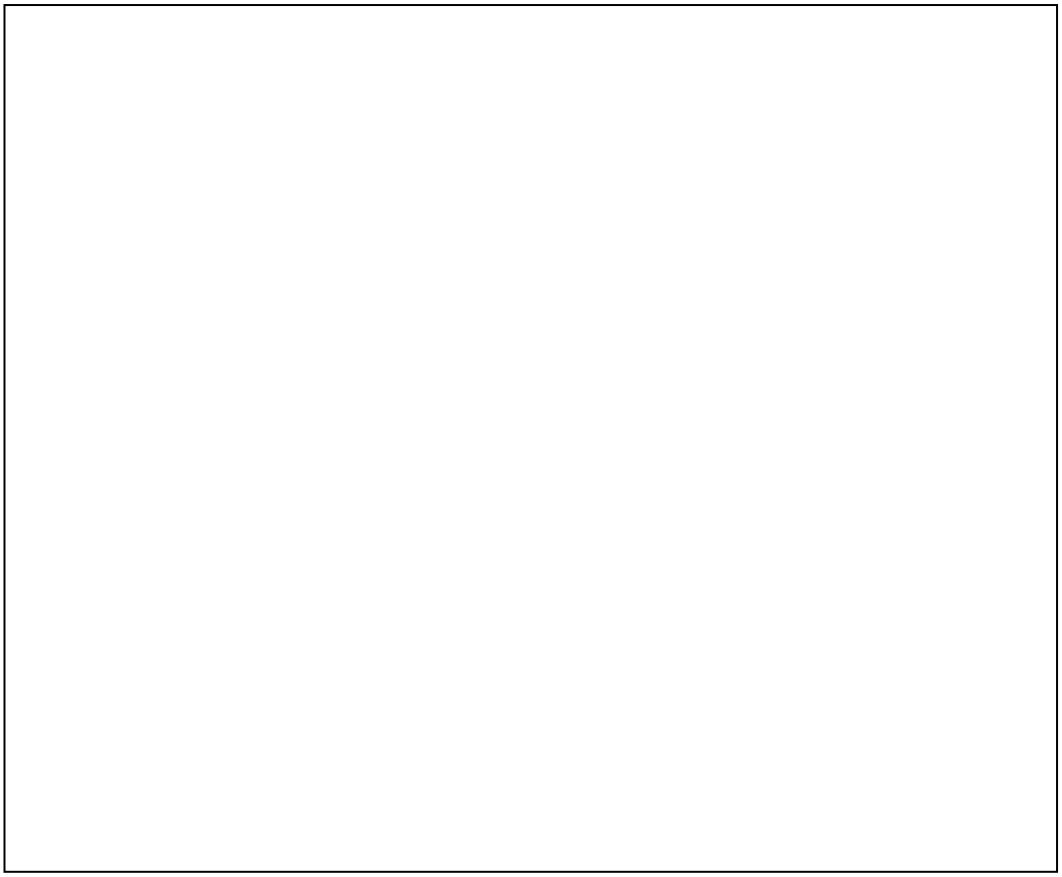
Stage 2: Instruction Decode

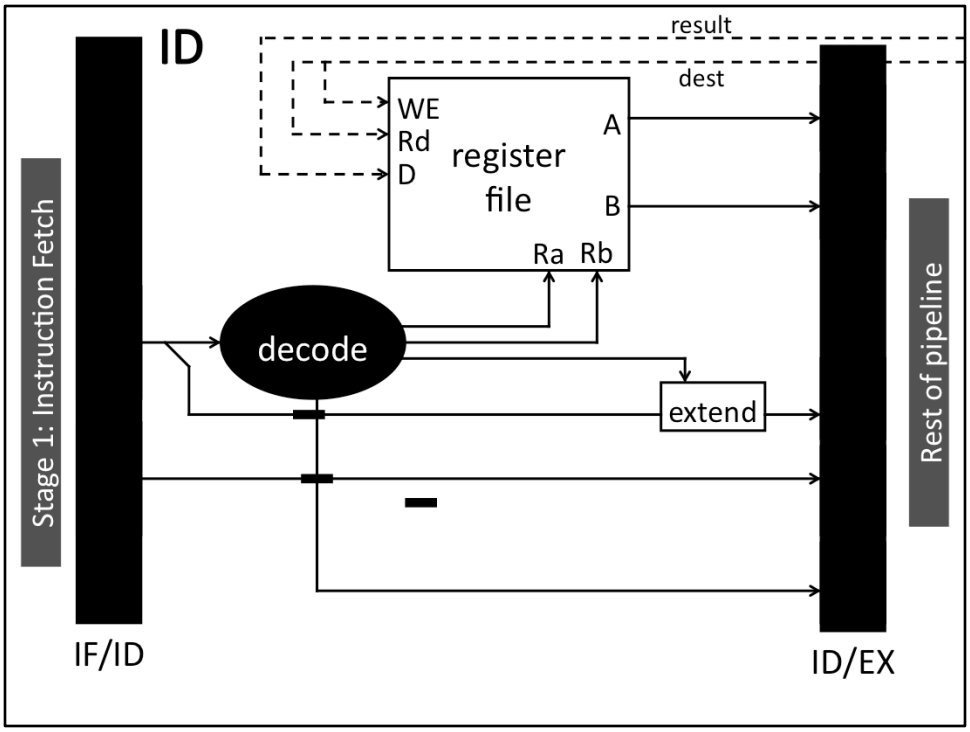
On every cycle:

- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

Write values of interest to pipeline register (ID/EX)

- Control information, Rd index, immediates, offsets, …
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)





Read in the first half, write in the second half. Later we will change this.

EX

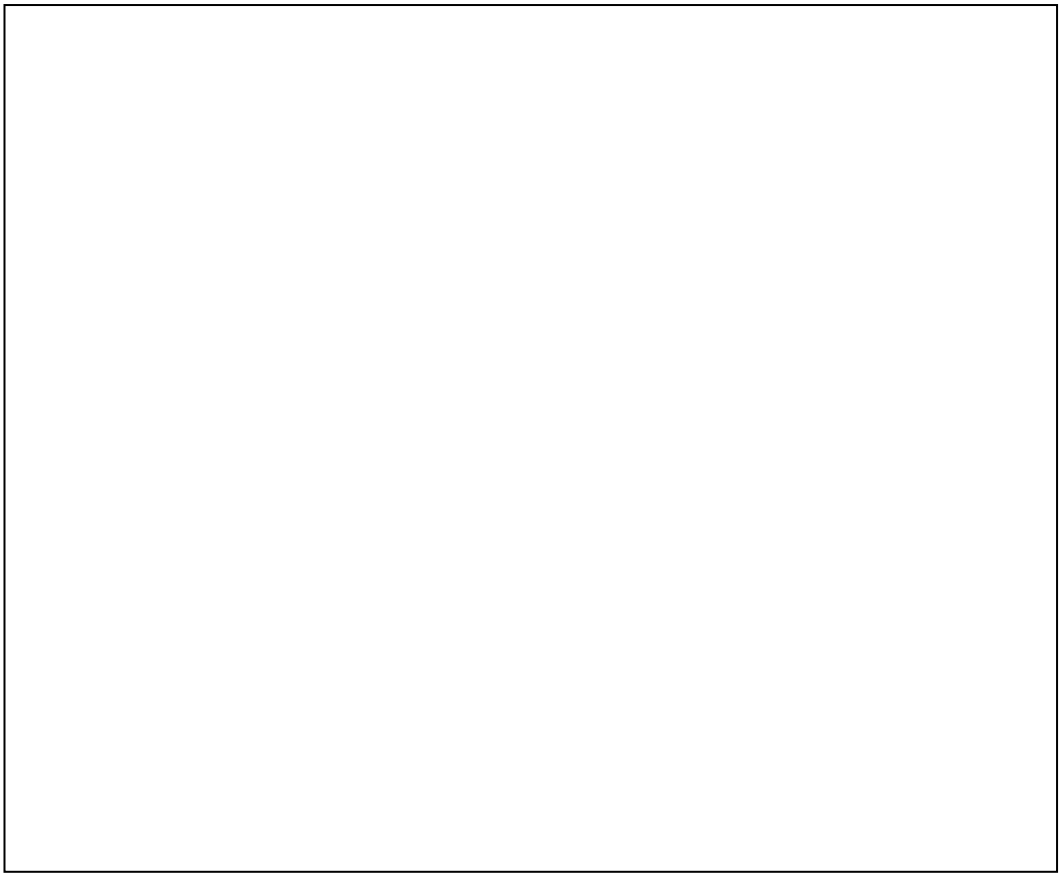
Stage 3: Execute

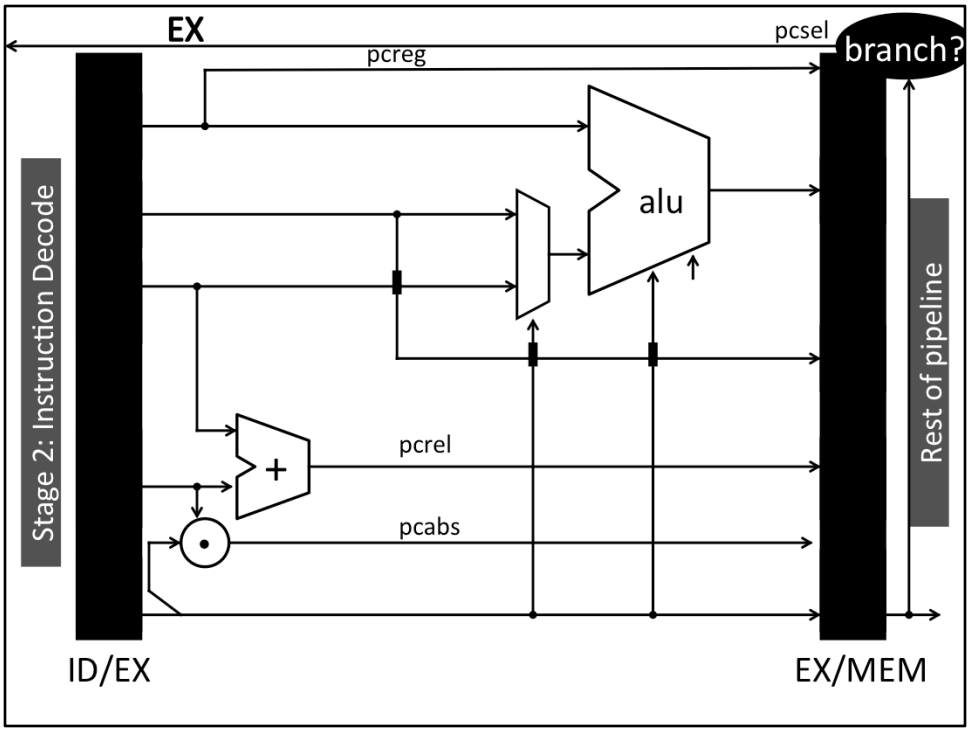
On every cycle:

- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)

- Control information, Rd index, …
- Result of ALU operation
- Value *in case* this is a memory store instruction





Pcreg: from a register j $s
Pcrel: from branch instructions, relative to the PC
Pcabs: absolute from j Label or jalr Label

In this case ctrl stores the target register
D is the final result

MEM

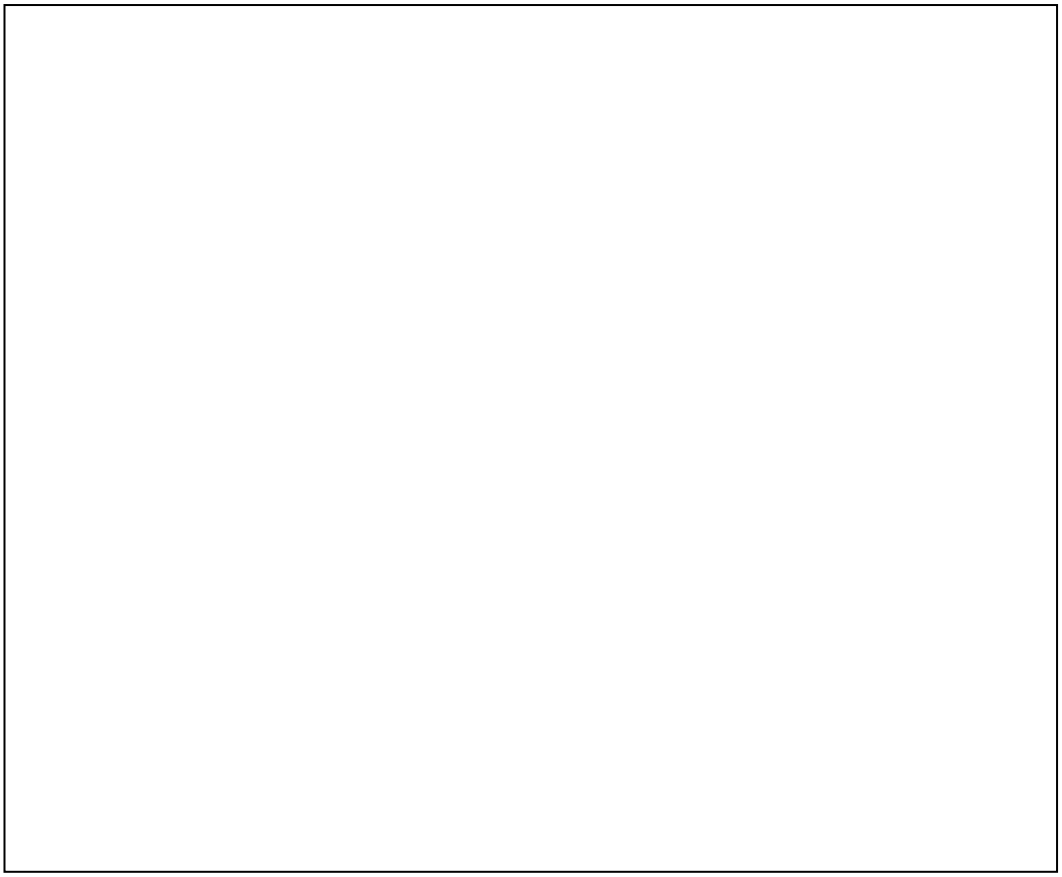
Stage 4: Memory

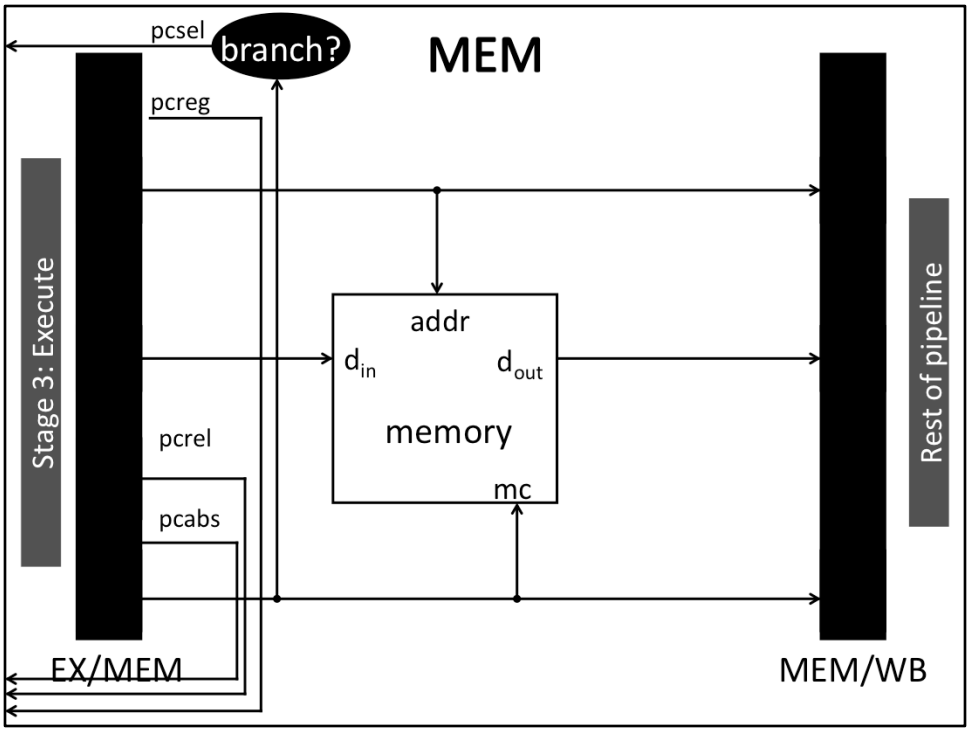
On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

Write values of interest to pipeline register (MEM/WB)

- Control information, Rd index, …
- Result of memory operation
- Pass result of ALU operation





From D you get sw or lw address in memory.

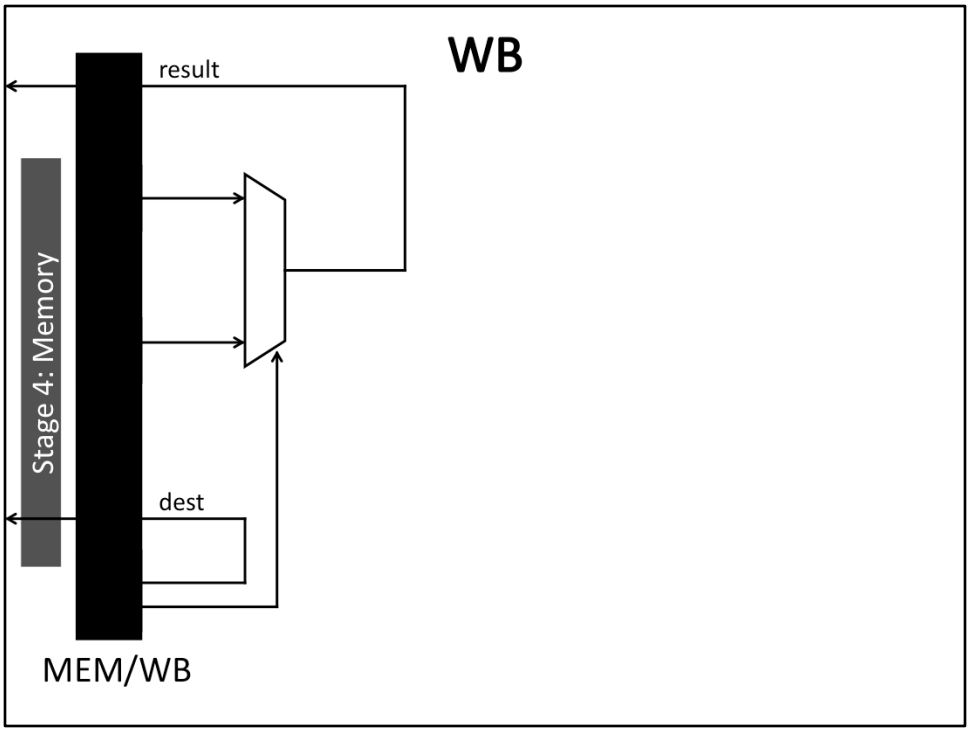
If sw, then B has the data to store.
D_out has the output lw result.

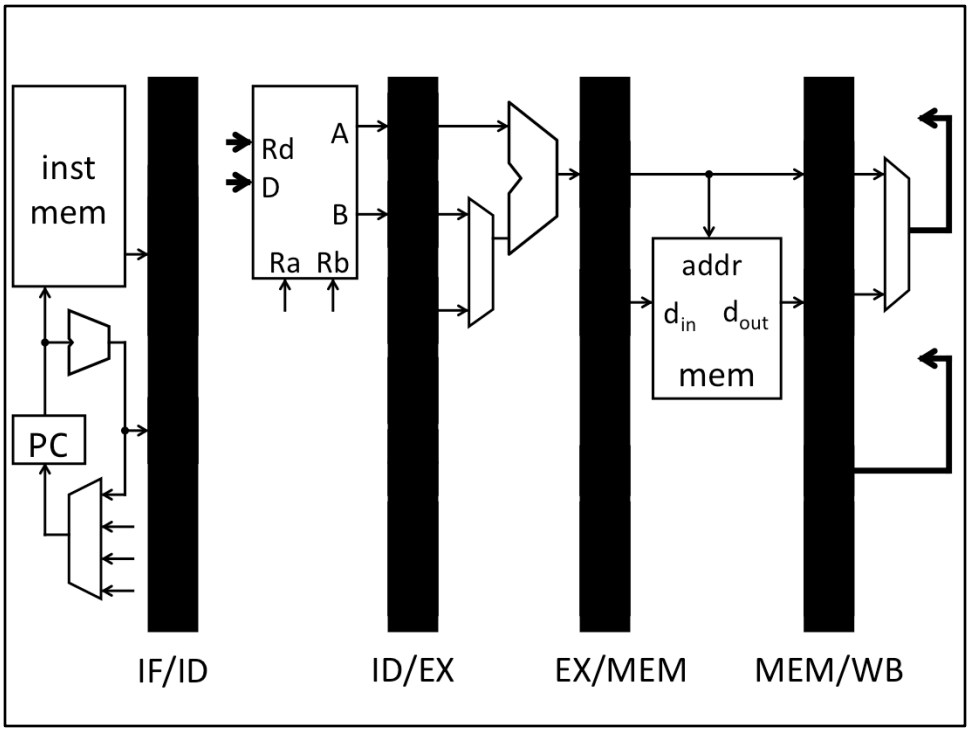
WB

Stage 5: Write-back

On every cycle:

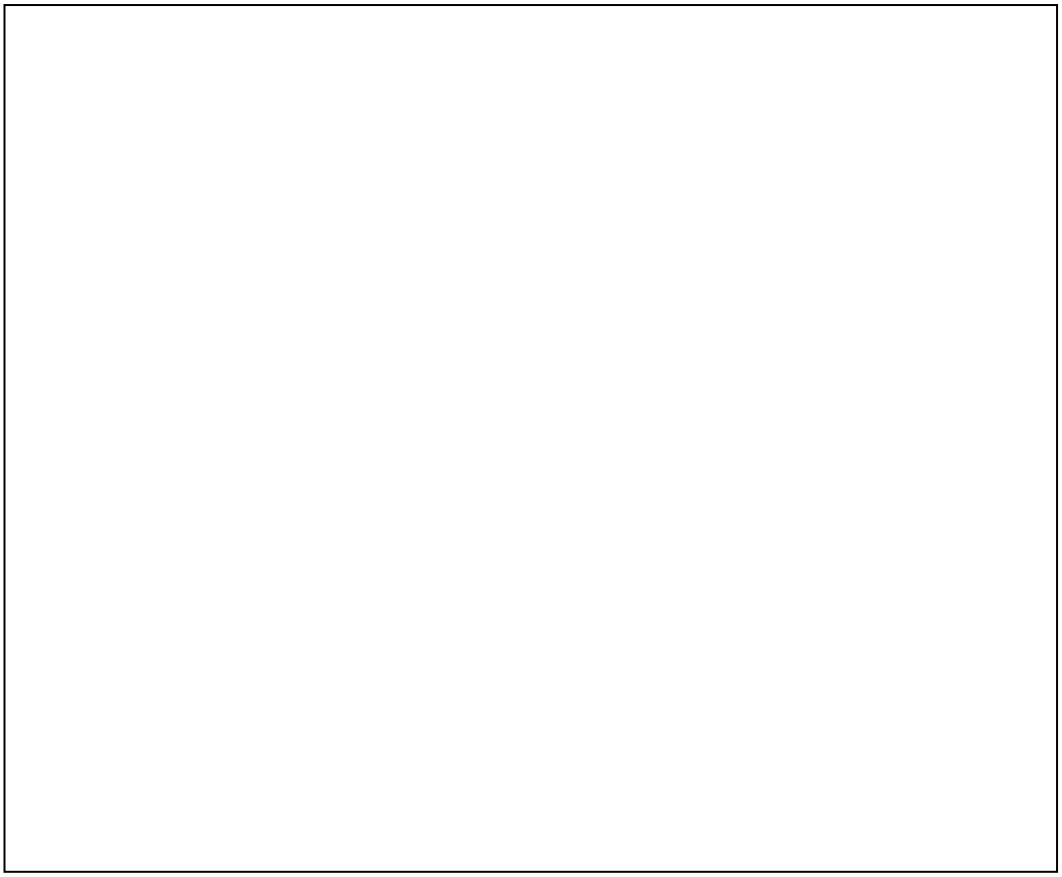
- Read MEM/WB pipeline register for values and control bits
- Select value and write to register file


WB
result
Stage 4: Memory
dest
MEM/WB


inst
mem
PC
Rd
D
Ra
Rb
A
B
addr
d_{in}
d_{out}
mem
IF/ID
ID/EX
EX/MEM
MEM/WB

Example: : Sample Code (Simple)

```
add    r3, r1, r2;
nand   r6, r4, r5;
lw     r4, 20(r2);
add    r5, r2, r5;
sw     r7, 12(r3);
```



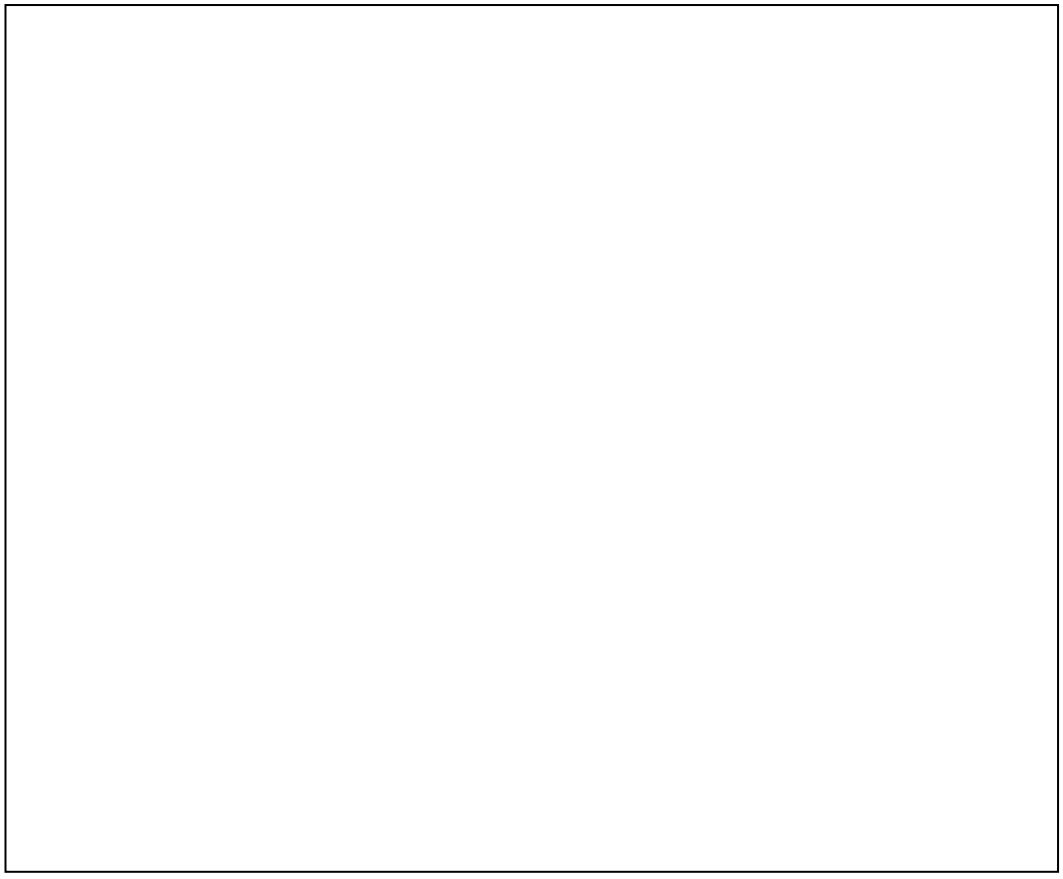
Example: Sample Code (Simple)

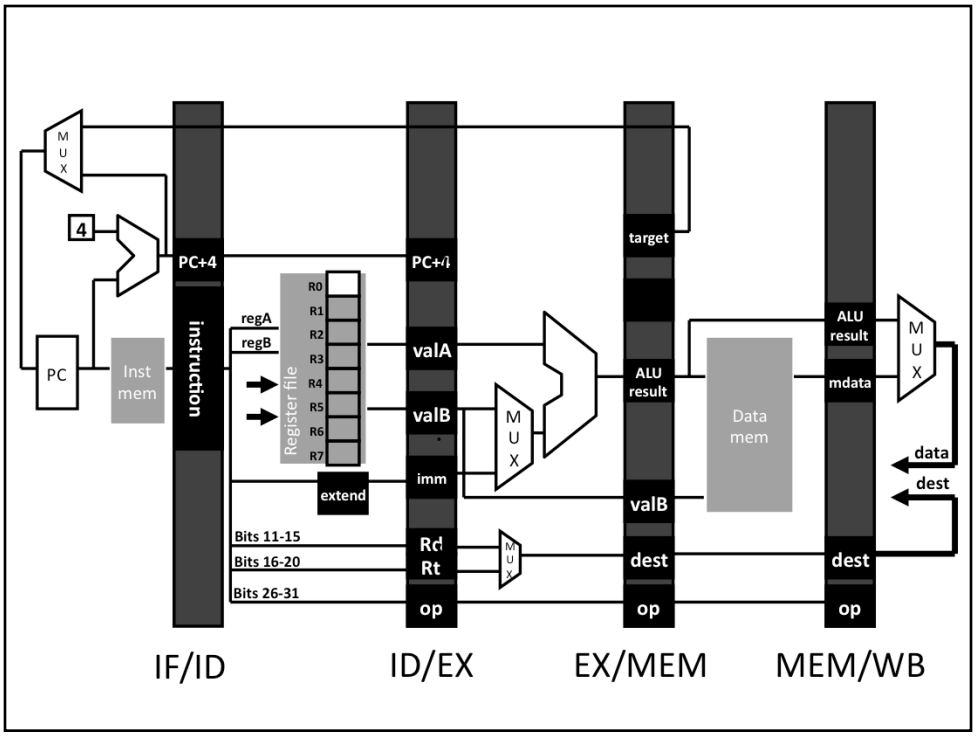
Assume eight-register machine

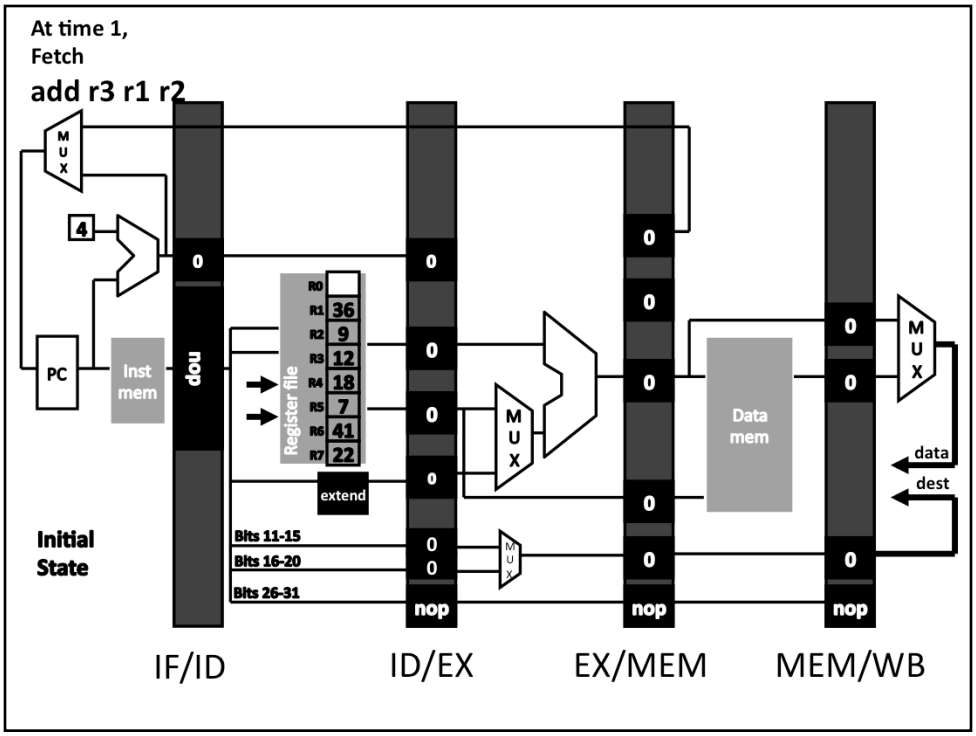
Run the following code on a pipelined datapath

```
add     r3  r1   r2   ;  reg 3 = reg 1 + reg 2
nand    r6   r4   r5   ;  reg 6 = ~(reg 4 & reg 5)
lw      r4   20 (r2)  ;  reg 4 =  Mem[reg2+20]
add     r5   r2   r5   ;  reg 5 = reg 2 + reg 5
sw      r7   12(r3)   ;  Mem[reg3+12] = reg 7
```

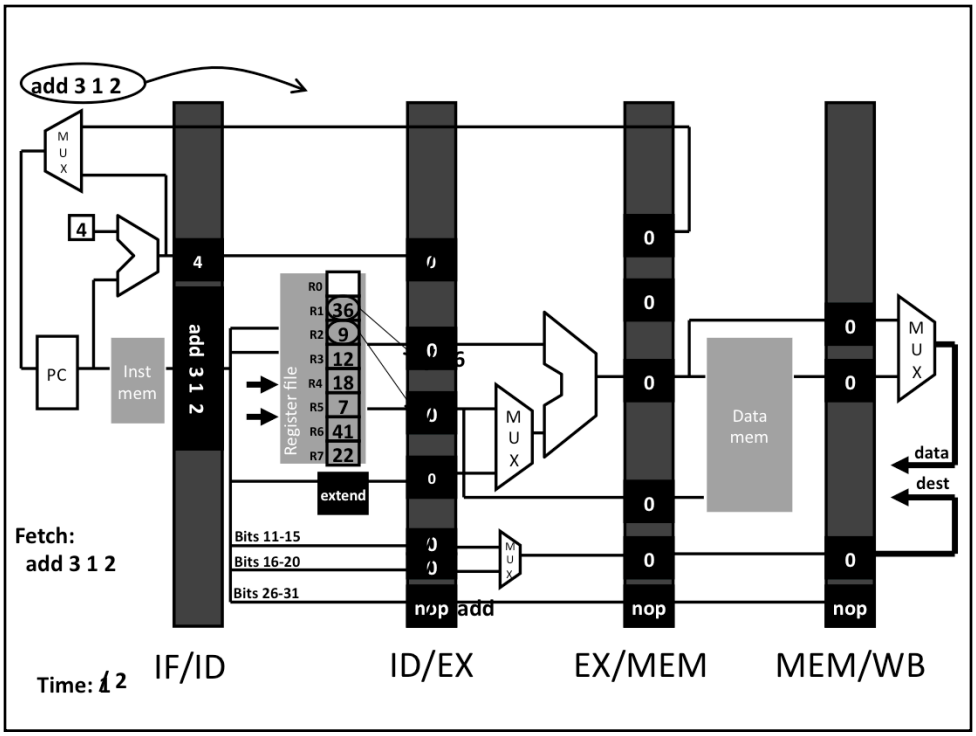
Slides thanks to Sally McKee

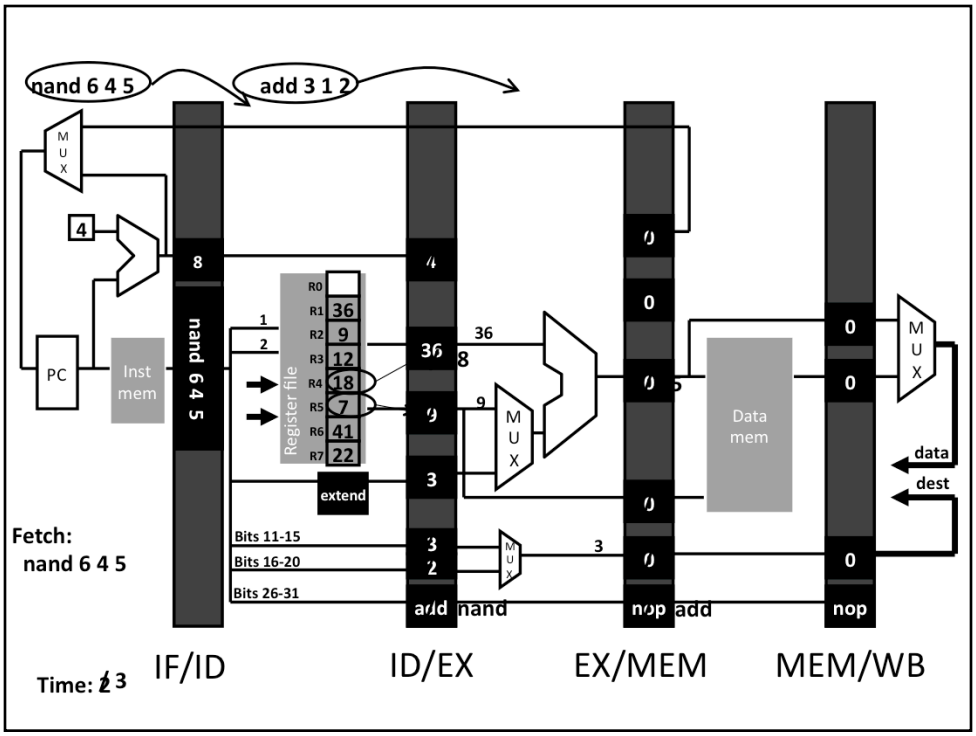


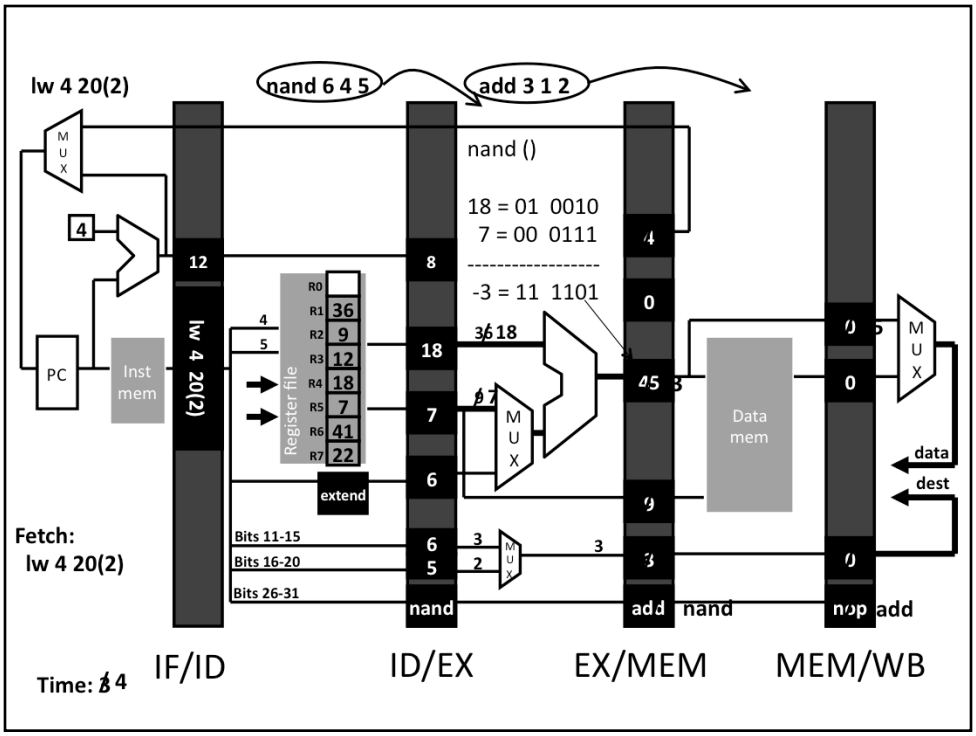

M
U
X
4
PC
Inst
mem
PC+4
instruction
regA
regB
Register file
R0
R1
R2
R3
R4
R5
R6
R7
extend
Bits 11-15
Bits 16-20
Bits 26-31
PC+4
valA
valB
imm
Rd
Rt
op
M
U
X
target
ALU
result
valB
dest
op
Data
mem
ALU
result
mdata
dest
op
M
U
X
data
dest
IF/ID
ID/EX
EX/MEM
MEM/WB



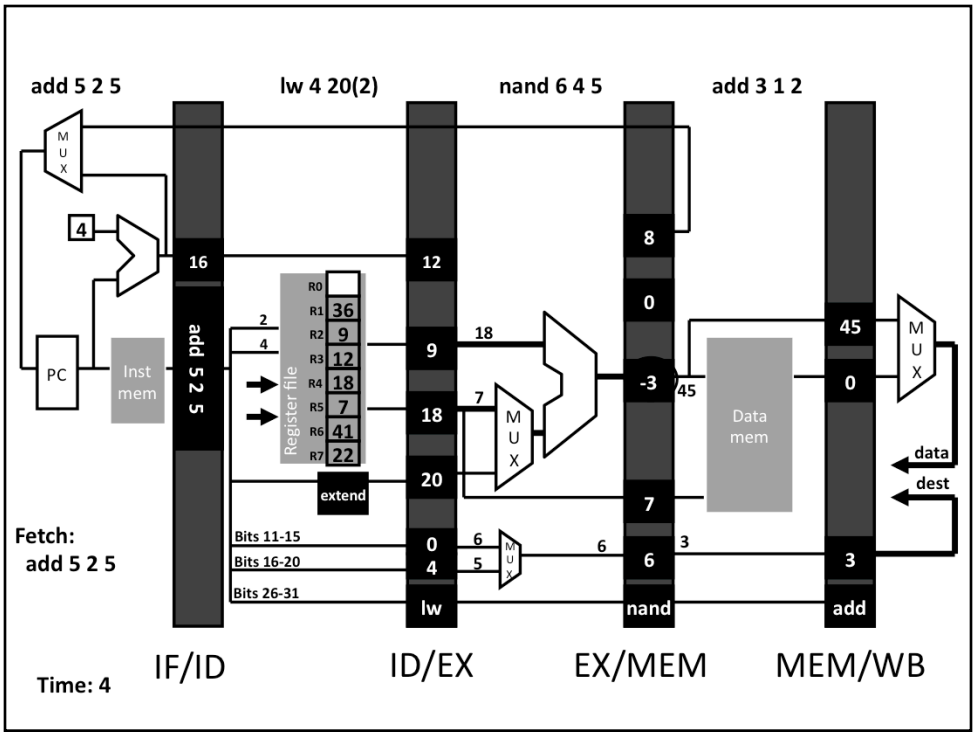
Nop is no operation.

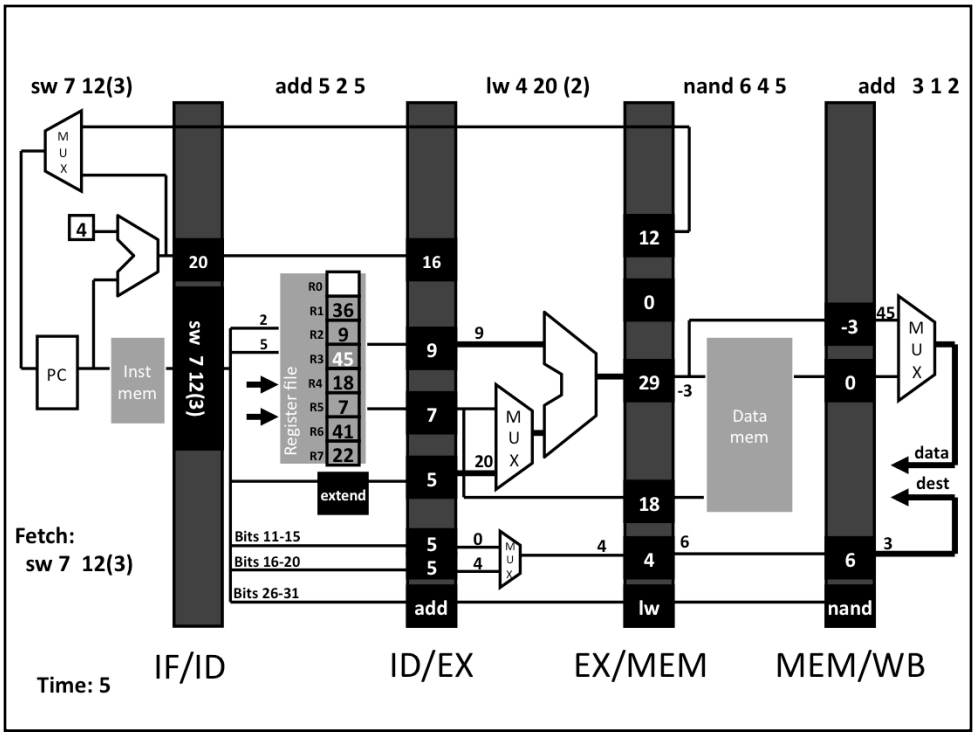

add 3 1 2
M U X
4
PC
Inst mem
add 3 1 2
Register file
R0
R1
R2
R3
R4
R5
R6
R7
36
9
12
18
7
41
22
extend
Bits 11-15
Bits 16-20
Bits 26-31
M U X
Data mem
data
dest
Fetch:
add 3 1 2
nop
add
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 1 2

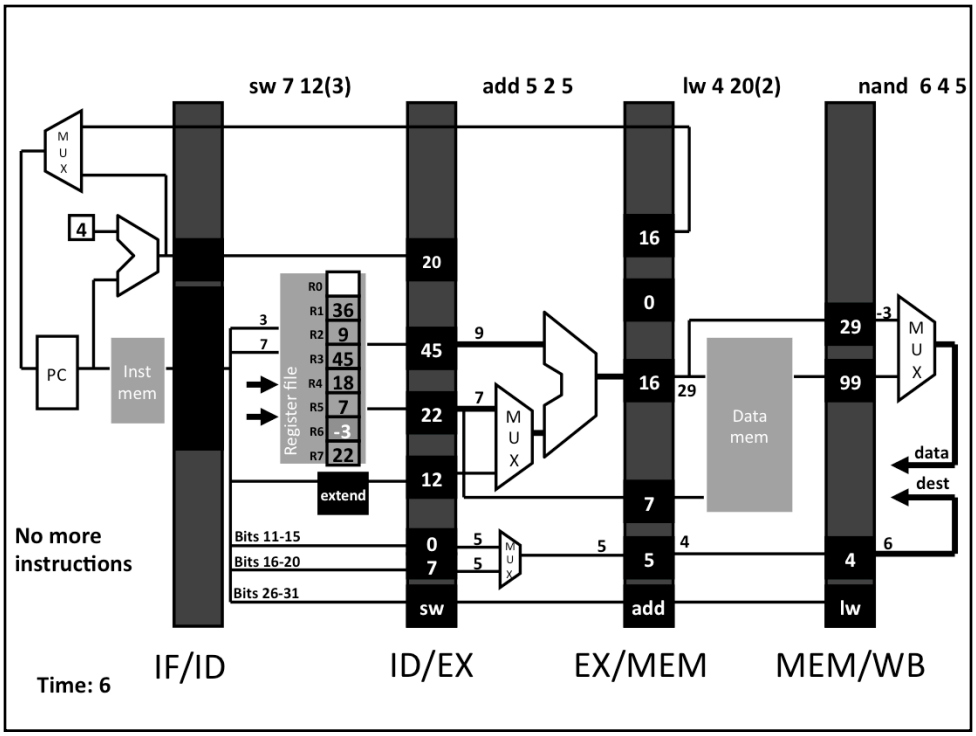

nand 6 4 5
add 3 1 2
MUX
4
8
PC
Inst mem
nand 6 4 5
Register file
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
1
2
extend
Bits 11-15
Bits 16-20
Bits 26-31
36
9
3
add
nand
nop
add
nop
Data mem
data
dest
Fetch:
nand 6 4 5
Time:
IF/ID
ID/EX
EX/MEM
MEM/WB

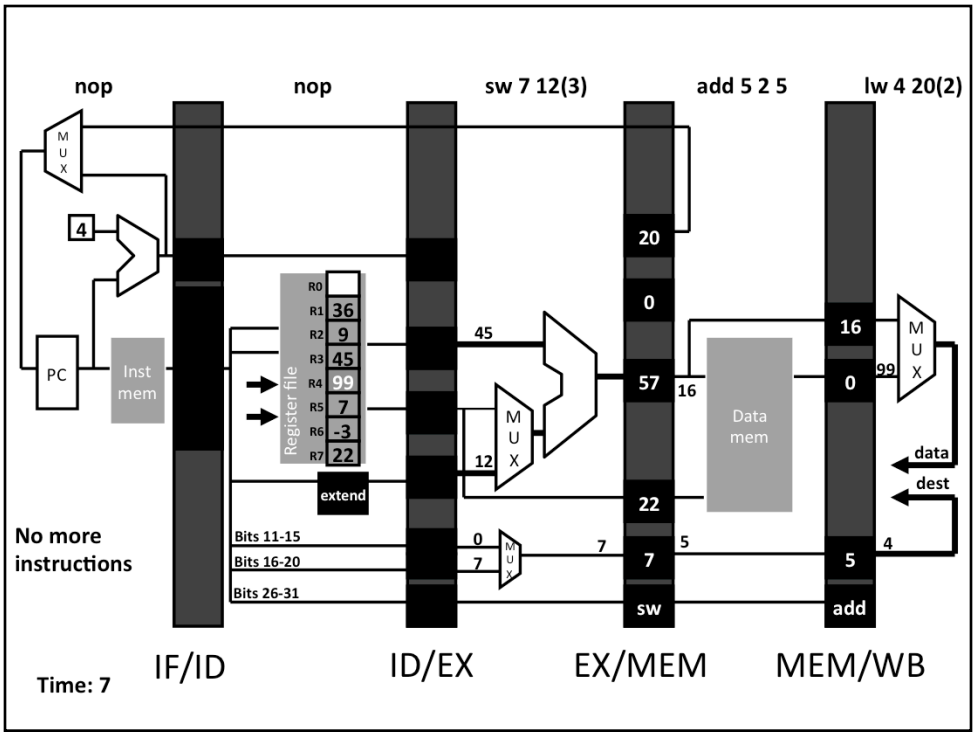


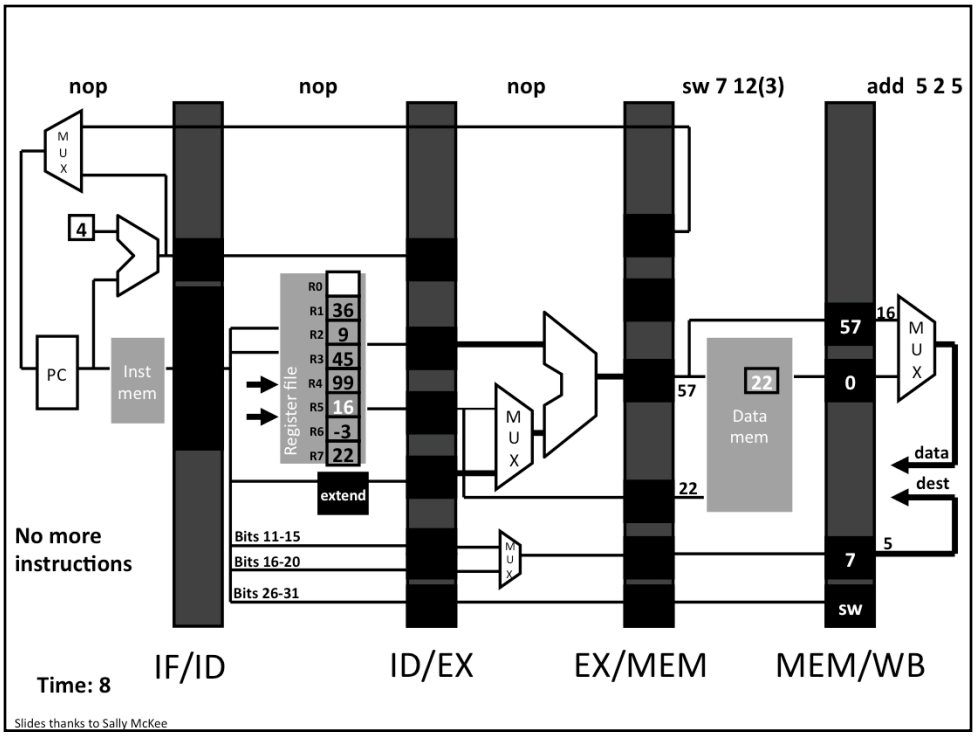
0x 00 0111
0x 10 0010
And = 10
Nand = 11111 1101

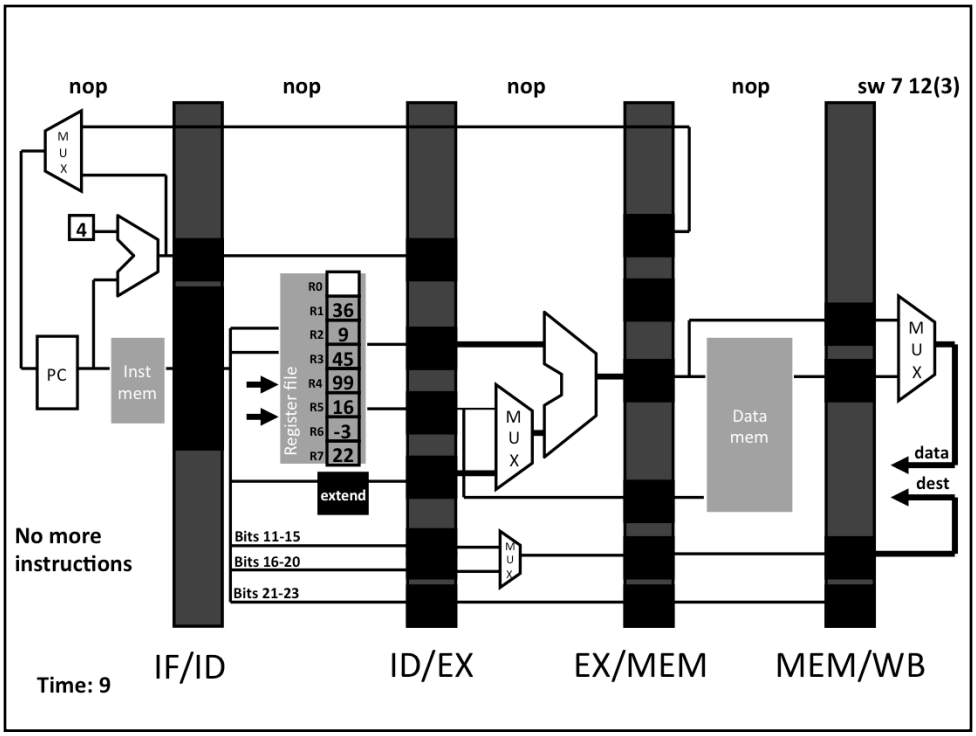

add 5 2 5
lw 4 20(2)
nand 6 4 5
add 3 1 2
M U X
4
16
PC
Inst mem
add 5 2 5
2
4
Register file
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
extend
Bits 11-15
Bits 16-20
Bits 26-31
12
9
18
20
0
4
lw
18
7
M U X
6
5
8
0
-3
7
6
nand
45
3
Data mem
45
0
3
add
M U X
data
dest
Fetch:
add 5 2 5
Time: 4
IF/ID
ID/EX
EX/MEM
MEM/WB


sw 7 12(3)
add 5 2 5
lw 4 20 (2)
nand 6 4 5
add 3 1 2
MUX
4
PC
Inst mem
20
sw 7 12(3)
2
5
Register file
R0
R1 36
R2 9
R3 45
R4 18
R5 7
R6 41
R7 22
extend
16
9
7
5
Bits 11-15
Bits 16-20
Bits 26-31
5
5
add
9
20
MUX
0
4
MUX
12
0
29
18
4
lw
-3
4
6
Data mem
-3
0
6
nand
45
MUX
3
data
dest
Fetch:
sw 7 12(3)
Time: 5
IF/ID
ID/EX
EX/MEM
MEM/WB


sw 7 12(3)
add 5 2 5
lw 4 20(2)
nand 6 4 5
MUX
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 18
R5 7
R6 -3
R7 22
3
7
20
45
22
12
0
7
sw
extend
Bits 11-15
Bits 16-20
Bits 26-31
9
7
5
5
16
0
16
7
5
add
29
4
Data mem
29
99
4
lw
-3
6
data
dest
No more instructions
Time: 6
IF/ID
ID/EX
EX/MEM
MEM/WB


nop
nop
sw 7 12(3)
add 5 2 5
lw 4 20(2)
MUX
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 99
R5 7
R6 -3
R7 22
extend
Bits 11-15
Bits 16-20
Bits 26-31
45
12
0
7
MUX
20
0
57
22
7
sw
16
Data mem
5
16
0
99
MUX
data
dest
4
add
No more instructions
Time: 7
IF/ID
ID/EX
EX/MEM
MEM/WB


nop
nop
nop
sw 7 12(3)
add 5 2 5
M U X
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 99
R5 16
R6 -3
R7 22
extend
Bits 11-15
Bits 16-20
Bits 26-31
M U X
57
22
Data mem
57
0
16
M U X
data
dest
5
7
sw
No more instructions
Time: 8
IF/ID
ID/EX
EX/MEM
MEM/WB
Slides thanks to Sally McKee


nop
nop
nop
nop
sw 7 12(3)
M
U
X
4
PC
Inst
mem
R0
R1 36
R2 9
R3 45
R4 99
R5 16
R6 -3
R7 22
Register file
extend
Bits 11-15
Bits 16-20
Bits 21-23
M
U
X
Data
mem
M
U
X
data
dest
No more
instructions
Time: 9
IF/ID
ID/EX
EX/MEM
MEM/WB

Takeaway

Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)

TALK ABOUT GPUs: 800 deep pipelines

Hazards

See P&H Chapter: 4.7-4.8

Hazards

3 kinds

- Structural hazards
  - Multiple instructions want to use same unit
- Data hazards
  - Results of instruction needed before
- Control hazards
  - Don't know which side of branch to take

1) Structural: say only 1 memory for instruction read and the MEM stage. That is a structural hazard. We have designed the MIPS ISA and our implementation of separate memory for instruction and data to avoid this problem. MIPS is designed very carefully to not have any structural hazards.

2) Data hazards arise when data needed by one instruction has not yet been computed (because it is further down the pipeline). We need a solution for this.

3) Control hazards arise when we don't know what the PC will be. This makes it not possible to push the next instruction down the pipeline. Till we know what the issue is.

Data Hazards

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. add r3, r1, r2

sub r5, r3, r4

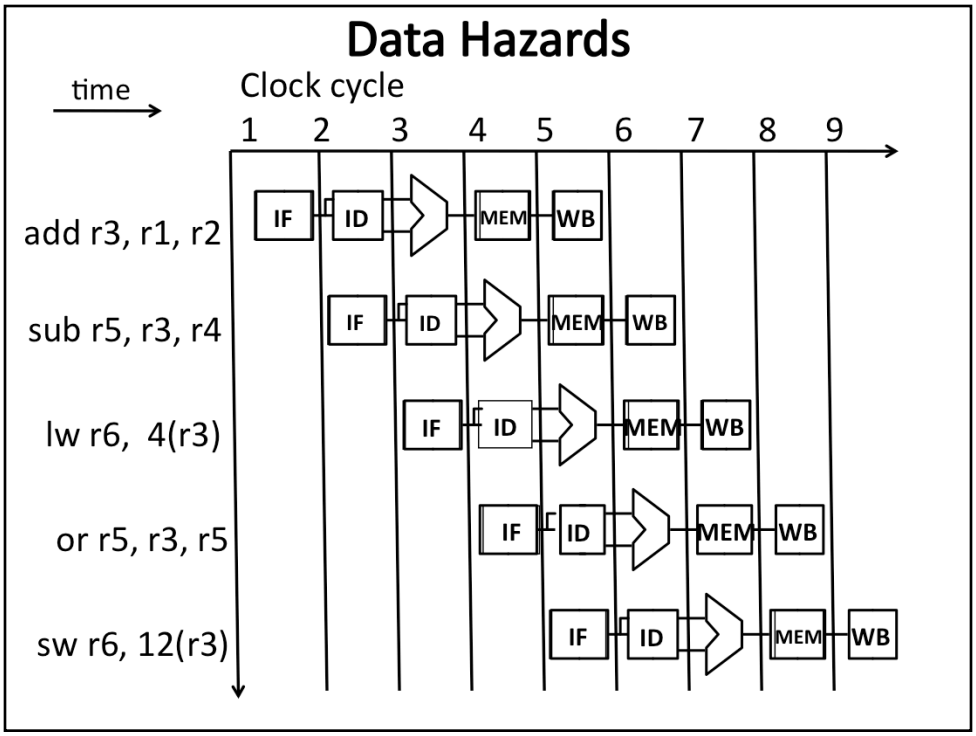
Need to detect and then fix such hazards

Why do data hazards occur?

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- instruction may read (need) values that are being computed further down the pipeline
  - In fact this is quite common

assumes (WE=0 implies rD=0) everywhere, similar for rA needed, rB needed (can ensure this in ID stage)



#1 and #2
#1 and #3
#1 and #4 (r3)

#2 and #4
#3 and #5

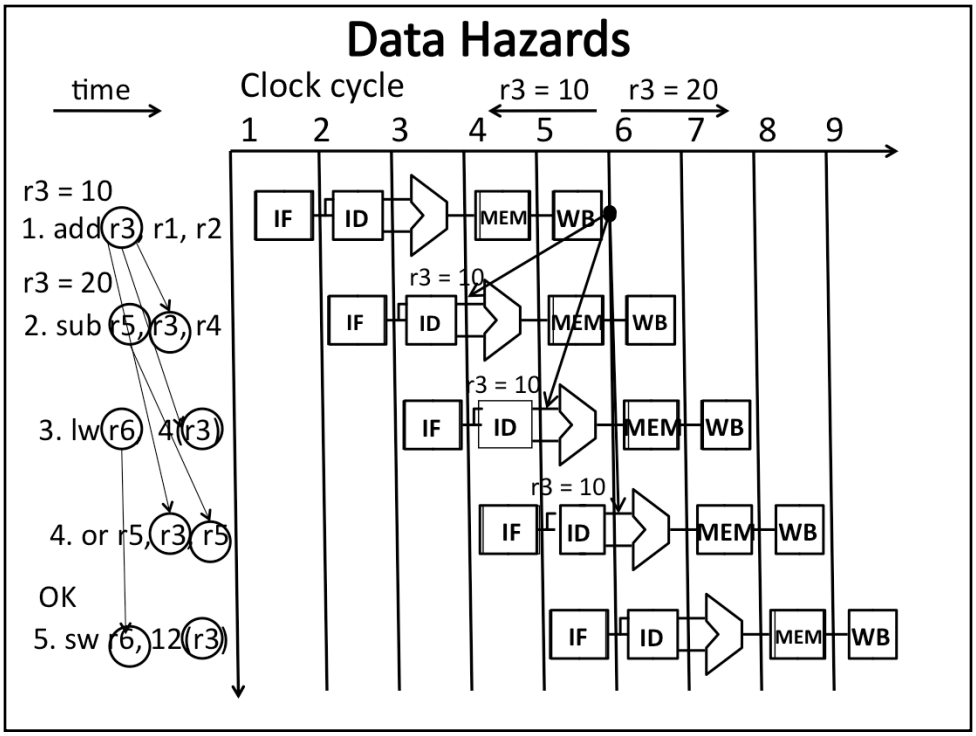
iClicker

```
add r3, r1, r2
sub r5, r3, r4
 lw r6,  4(r3)
  or r5, r3, r5
 sw r6, 12(r3)
```

How many data hazards due to r3 only

A) 1
B) 2
C) 3
D) 4
E) 5

C



#1 and #2
#1 and #3
#1 and #4 (r3) Can't go backward in time.

#2 and #4
#3 and #5

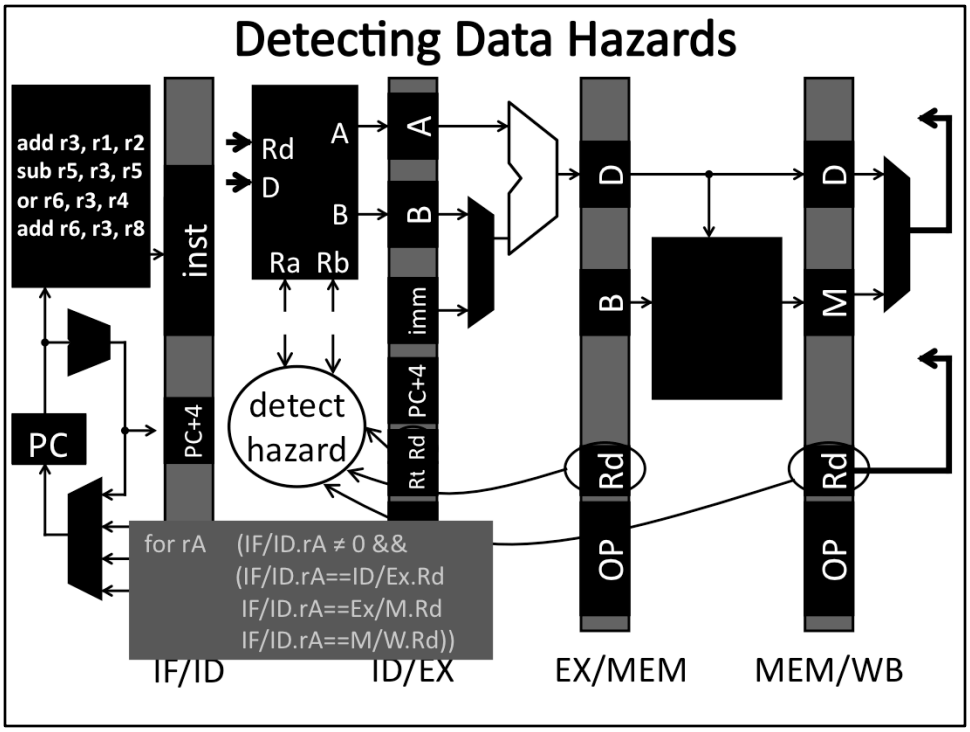
Data Hazards

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. add r3, r1, r2

sub r5, r3, r4

How to detect?


Detecting Data Hazards
add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8
inst
PC+4
PC
Rd
D
A
B
Ra
Rb
A
B
imm
PC+4
Rt
Rd
detect hazard
D
B
OP
Rd
D
M
OP
Rd
for rA (IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))
IF/ID
ID/EX
EX/MEM
MEM/WB

Detecting Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written
  - In fact this is quite common

How to detect?

```
(IF/ID.Ra != 0 &&
    (IF/ID.Ra == ID/EX.Rd ||
     IF/ID.Ra == EX/M.Rd ||
     IF/ID.Ra == M/WB.Rd))
     || (same for Rb)
```

assumes (WE=0 implies rD=0) everywhere, similar for rA needed, rB needed (can ensure this in ID stage)

Next Goal

- What to do if data hazard detected?
- Options
  - Nothing
    - Change the ISA to match implementation
  - Stall
    - Pause current and subsequent instructions till safe
  - Forward/bypass
    - Forward data value to where it is needed

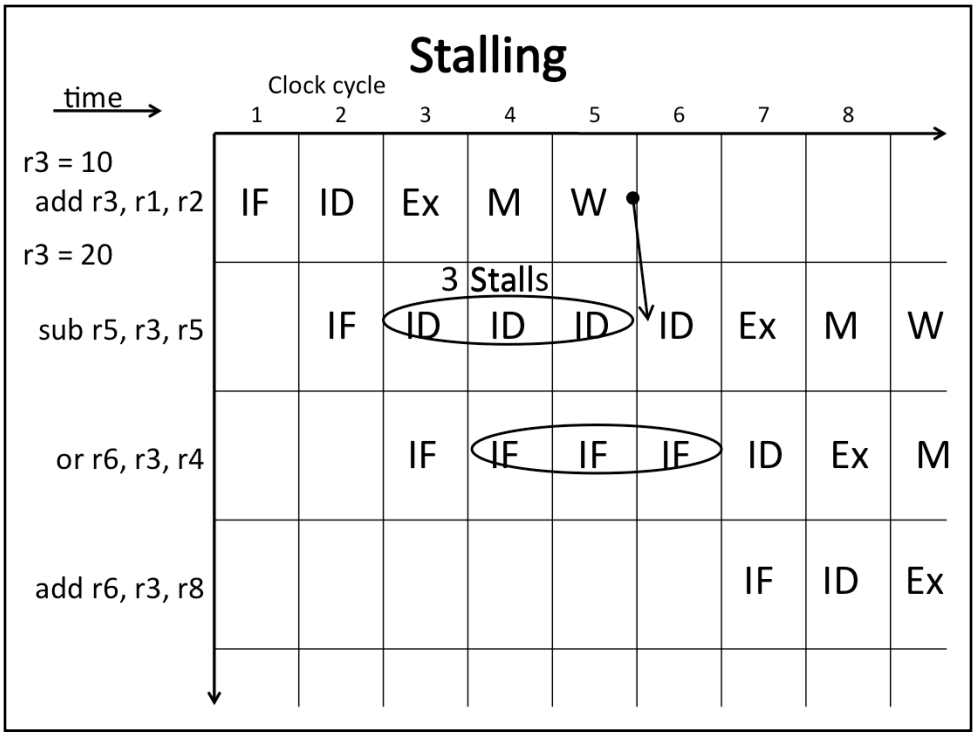
Nothing: Modify ISA to match implementation (not great as an option, violates abstraction. Requires restructuring by application. Might not always be possible).
Stall: Pause current and all subsequent instruction
Forward/Bypass: Steal correct value from elsewhere in pipeline

Stalling

How to stall an instruction in ID stage

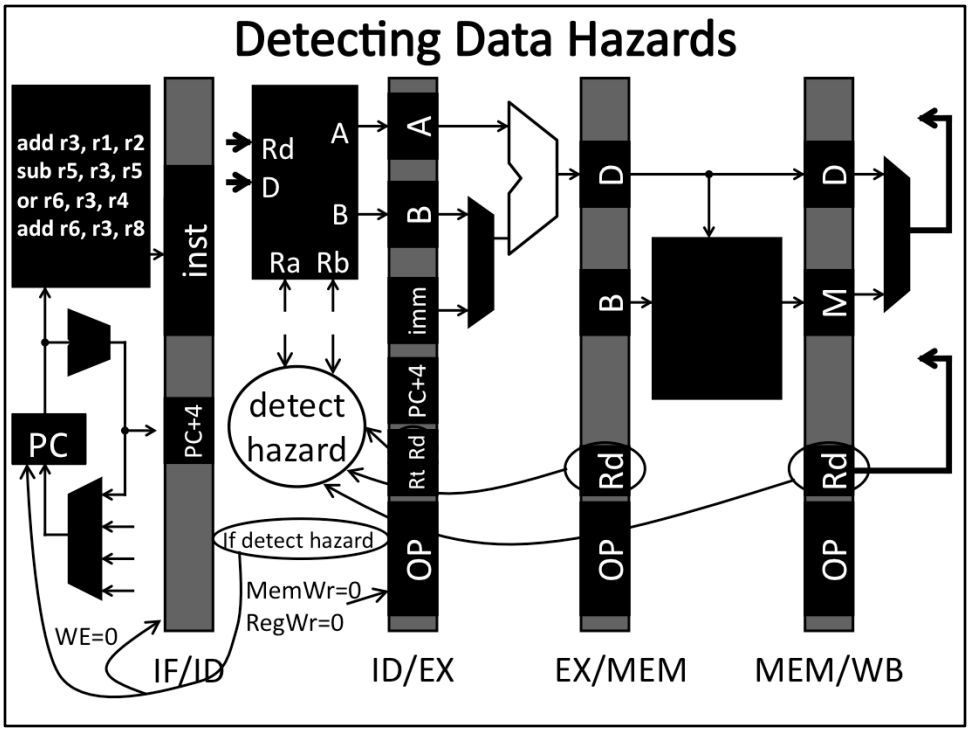
- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
  - innocuous "bubble" passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction

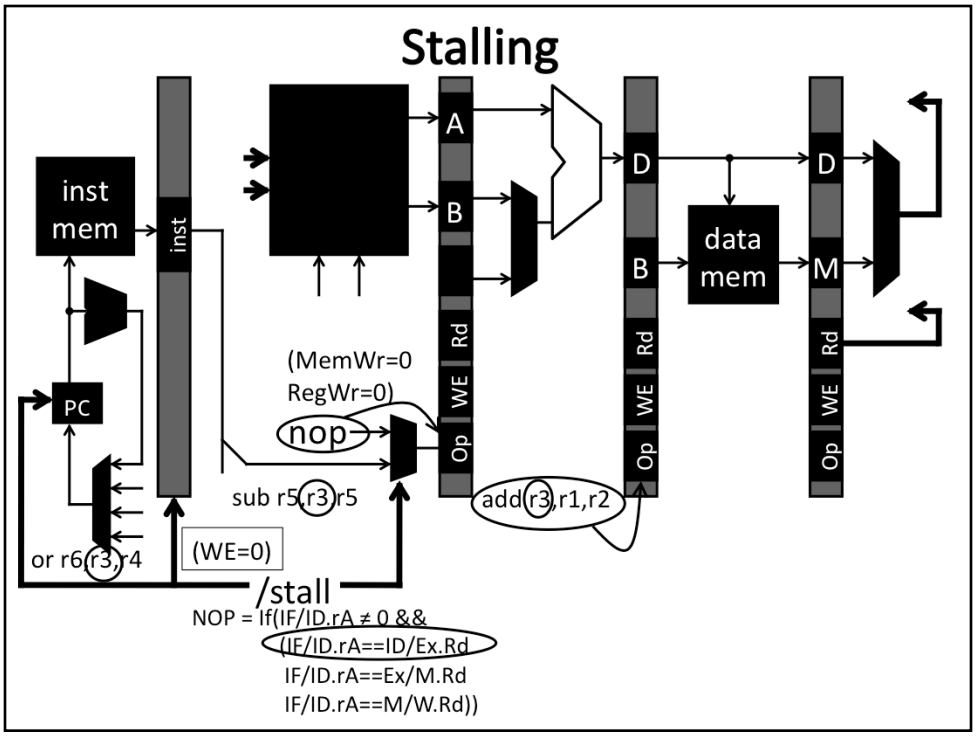
prev slide:
Q: what happens if branch in EX?
A: bad stuff: we miss the branch
So: lets try not to stall



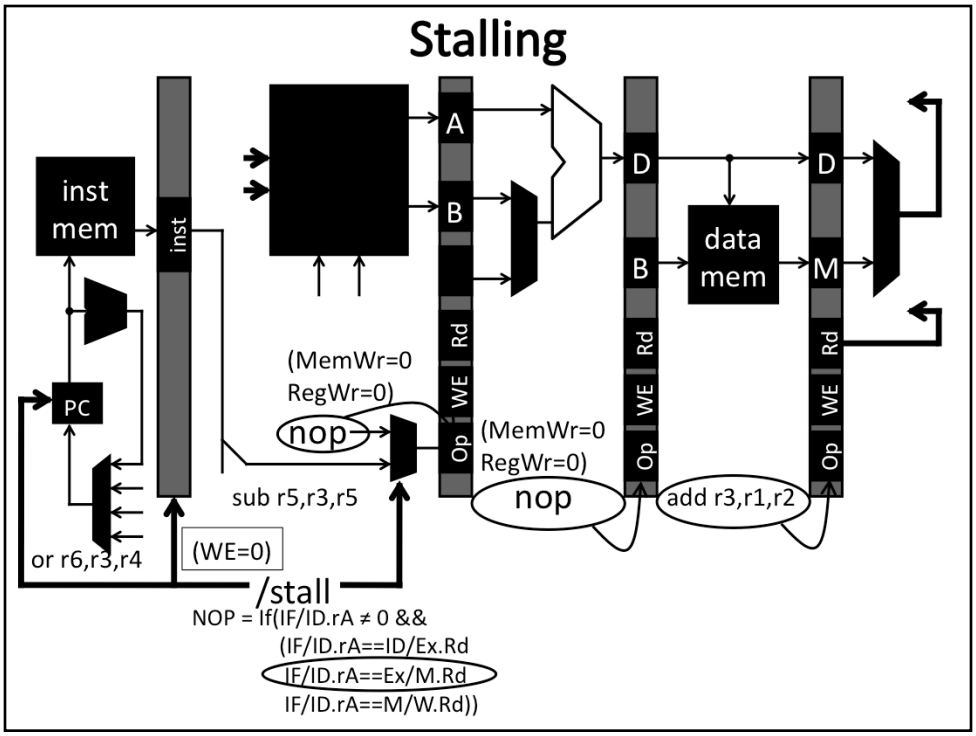
find hazards first, then draw

IMPORTANT: Arrows can only go forward in time.

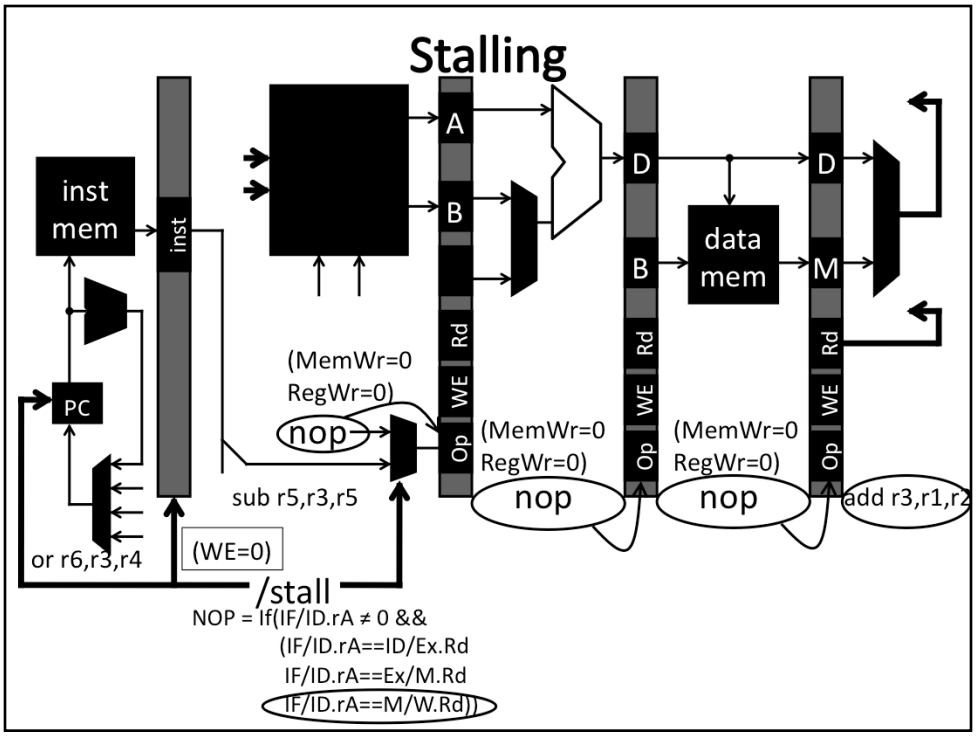

Detecting Data Hazards
add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8
inst
PC+4
PC
Rd
D
A
B
Ra
Rb
imm
PC+4
Rt
Rd
detect hazard
If detect hazard
MemWr=0
RegWr=0
WE=0
OP
A
B
D
B
Rd
OP
D
M
Rd
OP
IF/ID
ID/EX
EX/MEM
MEM/WB



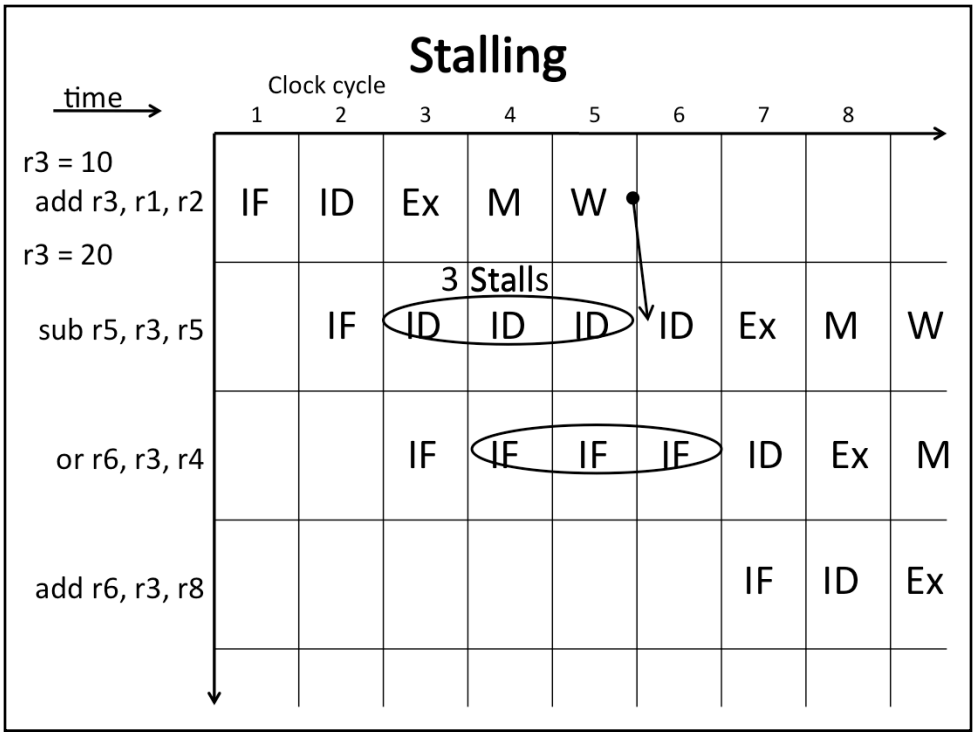
draw control signals for /stall
Q: what happens if branch in EX?
A: bad stuff: we miss the branch
So: lets try not to stall



draw control signals for /stall
Q: what happens if branch in EX?
A: bad stuff: we miss the branch
So: lets try not to stall



draw control signals for /stall
Q: what happens if branch in EX?
A: bad stuff: we miss the branch
So: lets try not to stall



find hazards first, then draw

IMPORTANT: Arrows can only go forward in time.

Stalling

How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
  - innocuous "bubble" passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction

prev slide:
Q: what happens if branch in EX?
A: bad stuff: we miss the branch
So: lets try not to stall

Takeaway

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards.

Stalling introduces NOPs ("bubbles") into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles in pipeline significantly decrease performance.